MPI on the Grid

William Gropp
Mathematics and Computer Science
Argonne National Laboratory
http://www.mcs.anl.gov/~gropp
With help from
Rusty Lusk, Nick Karonis

LANS

Outline

- Why MPI?
- MPI Point-to-point communication
- MPI Collective communication
- Performance and Debugging
- MPICHG2

Why use MPI on the Grid?

- Applications already exist
- Tools exist
  - ♦ Libraries and components, some enabled for the Grid
    - E.g., Cactus (Gordon Bell winner)
- Simplifies Development
  - ♦ “Build locally, run globally”
  - ♦ NSF TeraGrid plans this approach

Why use the MPI API on the Grid?

- MPI’s *design* is latency tolerant
- Separation of concerns matches the needs of Grid infrastructure
  - ♦ MPI itself has no “grid awareness”
  - ♦ MPI designed to operate in many environments
    - The Grid is “just” another such environment

Specification and Implementation

- MPI is a specification, not a particular implementation
- MPI (the specification) has many features that make it well-suited for Grid computing
- There are many opportunities for enhancing MPI implementations for use in Grid computing
  - Some tools already exist
  - Great opportunities for research and papers for EuroPVMMPI'03!

Grid Issues

- Simplest "grid" model
  - Multiple systems that do not share all resources
    - Example: Several clusters, each with its own file system
- More complex model
  - Multiple systems in separate administrative domains
    - Example: Several clusters, each administered by a different organization, with different access policies and resources
- (Mostly) Shared Properties
  - Geographic separation
    - From 1 to 10000km
    - Each 1000km gives at least 3ms of latency
      - Typical of *disk* access!
  - Process management separate from communication
    - Must not assume any particular mechanism for creating processes

Issues When Programming for the Grid

- Latency
  - Using MPI's send modes to hide latency
- Hierarchical Structure
  - Developing and using collective communication for high, unequal latency
- Handling Faults

University of Chicago Department of Energy

Quick review of MPI Message passing

- Basic terms
  - nonblocking - Operation does not wait for completion
  - synchronous - Completion of send *requires* initiation (but not completion) of receive
  - ready - *Correct* send requires a matching receive
  - asynchronous - communication and computation take place simultaneously, ***not*** an MPI concept (implementations *may* use asynchronous methods)

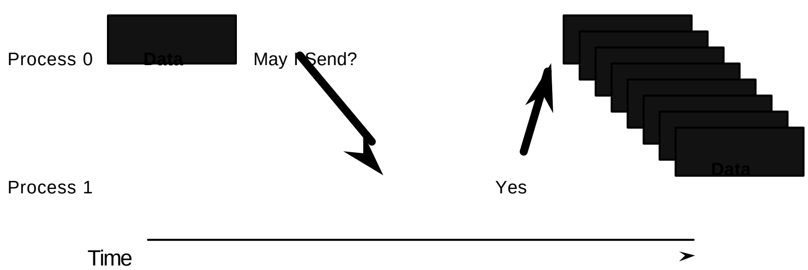
University of Chicago Department of Energy

Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**MPI_Ssend**): The send does not complete until a matching receive has begun.
  - Buffered mode (**MPI_Bsend**): The user supplies a buffer to the system for its use.
  - Ready mode (**MPI_Rsend**): User guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - Undefined behavior if matching receive not posted
- Non-blocking versions (**MPI_Issend**, etc.)
- **MPI_Recv** receives messages sent in any mode.

What is message passing?

- Data transfer plus synchronization

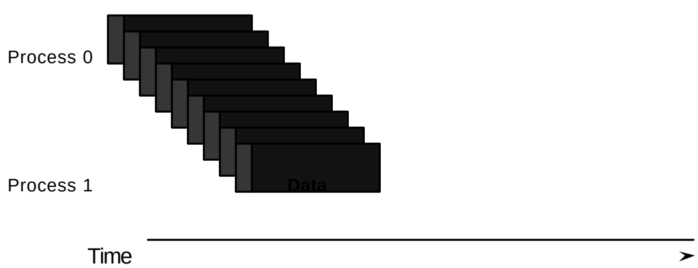


- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Message protocols

- Message consists of “envelope” and data
  - Envelope contains tag, communicator, length, source information, plus impl. private data
- Short
  - Message data (message for short) sent with envelope
- Eager
  - Message sent assuming destination can store
- Rendezvous
  - Message not sent until destination oks

Eager Protocol



- Data delivered to process 1
  - No matching receive may exist; process 1 must then buffer and copy.

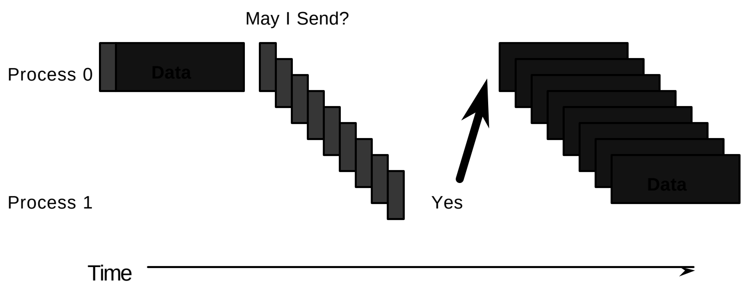
Eager Features

- Reduces synchronization delays
- Simplifies programming (just MPI_Send)
- Requires significant buffering
- May require active involvement of CPU to drain network at receiver's end
- May introduce additional copy (buffer to final destination)
- Minimizes latency

How Scaleable is Eager Delivery?

- Buffering must be reserved for arbitrary senders
- User-model mismatch (often expect buffering allocated entirely to "used" connections).
- Common approach in implementations is to provide same buffering for all members of MPI_COMM_WORLD; this is optimizing for non-scaleable computations
- Scalable implementations that exploit message patterns are possible (but not widely implemented)

Rendezvous Protocol



- Envelope delivered first
- Data delivered when user-buffer available
  - Only buffering of envelopes required

Rendezvous Features

- Robust and safe
  - (except for limit on the number of envelopes...)
- May remove copy (user to user direct)
- More complex programming (waits/tests)
- May introduce synchronization delays (waiting for receiver to ok send)
- Three-message handshake introduces latency

Short Protocol

- Data is part of the envelope
- Otherwise like eager protocol
- May be performance optimization in interconnection system for short messages, particularly for networks that send fixed-length packets (or cache lines)

Implementing MPI_Isend

- Simplest implementation is to always use rendezvous protocol:
  - MPI_Isend delivers a request-to-send control message to receiver
  - Receiving process responds with an ok-to-send
    - May or may not have matching MPI receive; only needs buffer space to store incoming message
  - Sending process transfers data
- Wait for MPI_Isend request
  - wait for ok-to-send message from receiver
  - wait for data transfer to be complete on sending side

Alternatives for MPI_Isend

- Use a short protocol for small messages
  - No need to exchange control messages
  - Need guaranteed (but small) buffer space on destination for short message envelope
  - Wait becomes a no-op
- Use eager protocol for modest sized messages
  - Still need guaranteed buffer space for both message envelope and eager data on destination
  - Avoids exchange of control messages

Implementing MPI_Send

- Can't use eager always because this could overwhelm the receiving process

```
if (rank != 0) MPI_Send( 100 MB of data )
else receive 100 MB from each process
```

- Would like to exploit the blocking nature (can wait for receive)
- Would like to be fast
- Select protocol based on message size (and perhaps available buffer space at destination)
  - Short and/or eager for small messages
  - Rendezvous for longer messages

Implementing MPI_Rsend

- Just use MPI_Send; no advantage for users
- Use eager always (or short if small)
  - Even for long messages

Choosing MPI Alternatives

- MPI offers may ways to accomplish the same task
- Which is best?
  - Just like everything else, it depends on the vendor, system architecture, computational grid environment
  - Like C and Fortran, MPI provides the programmer with the tools to achieve high performance without sacrificing portability

LANS

Using MPI_Rsend to Minimize Latency

- For high-latency environments, avoid message handshakes
  - ♦ Problem: Must guarantee that sufficient space is available at destination for the message *without* exchanging messages
  - ♦ Use *algorithmic features* and *double buffering* to enforce guarantees

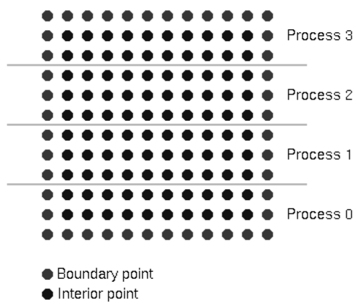
LANS

Using MPI_Rsend

- Illustrate with simple Jacobi example
  - ♦ Typical data motion for many applications
  - ♦ Specific *numeric* algorithm is obsolete

Jacobi Iteration

- Simple parallel data structure



- Processes exchange rows with neighbors

Send and Recv

- Simplest use of send and recv
- ```
  MPI_Status status;
  MPI_Send( xlocal+m*lrow, m, MPI_DOUBLE, up_nbr, 0,
  comm );
  MPI_Recv( xlocal, m, MPI_DOUBLE, down_nbr, 0,
  comm, &status );
  MPI_Send( xlocal+m, m, MPI_DOUBLE, down_nbr, 0,
  comm);
  MPI_Recv( xlocal+m*(lrow+1), m, MPI_DOUBLE,
  up_nbr, 1, comm, &status);
  ```
- Receives into ghost rows
  ```

LANS

What is the whole algorithm?

1. Loop
   1. Exchange ghost cells
   2. Perform local computation (Jacobi sweep)
   3. Compute convergence test using MPI_Allreduce
2. Until converged

LANS

What is the ready-send version of the algorithm?

1. Initialize (post nonblocking receives, barrier or initial MPI_Allreduce)
2. Loop
   1. Exchange ghost cells using MPI_Rsend (or MPI_Irsend)
   2. Perform local computation (Jacobi sweep)
   3. Post nonblocking receives for next iteration
   4. Compute convergence test using MPI_Allreduce
3. Until converged
   1. Cancel unneeded receives with MPI_Cancel

Rsend and Irecv

- void init_exchng(...)
```
{
   MPI_Irecv( xlocal, m, MPI_DOUBLE, down_nbr, 0,
              comm, &m->req[0]);
   MPI_Irecv( xlocal+m*(lrow+1), m, MPI_DOUBLE,
              up_nbr, 1, comm, &m->req[1]);
}
```
- void do_exchng(...)
```
{
   MPI_Rsend( xlocal+m*lrow, m, MPI_DOUBLE, up_nbr, 0,
               comm );
   MPI_Rsend( xlocal+m, m, MPI_DOUBLE, down_nbr, 0, comm);
   MPI_Waitall( 2, m->req, MPI_STATUSES_NULL );
}
```
- Void clear_exchng(...)
```
{
   MPI_Cancel( m->req[0] ); MPI_Cancel( m->req[1] );
}
```


Recommendations

- Aggregate short messages
- Structure algorithm to use MPI_Rsend or MPI_Irsend
- Avoid MPI_Ssend
- Once more MPI implementations support MPI_THREAD_MULTIPLE, restructure algorithms to place MPI communication into a separate thread
  - MPI_Init_thread is used to request a particular level of thread support; it returns as a parameter the available level of thread support

Research Topics

- Reduce the number of internal messages for sending large messages
  - "Receiver rendezvous" instead of sender rendezvous
    - Difficulties with MPI_ANY_SOURCE might be addressed with communicator-specific attribute values
  - Adaptive allocation of buffer space (increasing eager threshold), to make Rsend approach unnecessary
  - "Infinite window" replacements for IP/TCP
    - Provide effect of multiple TCP paths but with sensible flow control and fair resource sharing

Using MPI Collective Operations

- Collective routines offer a simpler programming model
- Puts the burden of implementing the best communication algorithm on the MPI implementor
  - Typical implementations not optimized (see Tuesday's talk on MPICH2)
  - Few implementations are *grid* optimized
- I will discuss the implementation in MPICH-G2
  - Another good implementation is MagPIE (see http://www.cs.vu.nl/albatross/#software)

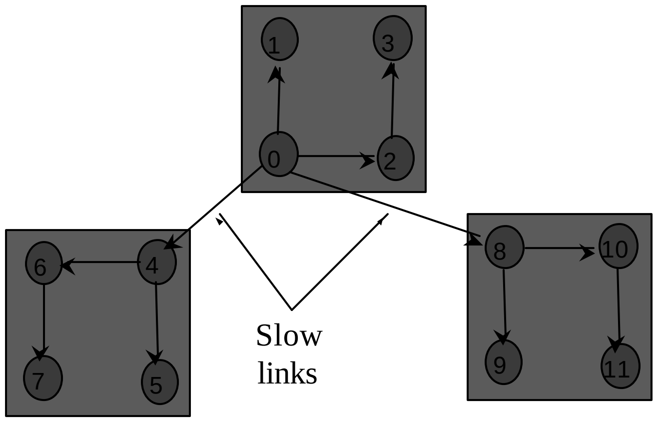
Topology-Aware Collective Team

- Original Design and Implementation
  - Bronis R. de Supinski
  - Nicholas T. Karonis
  - Ian Foster
  - William Gropp
  - Ewing Lusk
  - John Bresnahan
- Updated Implementation by
  - Sebastien Lacour

Multi-level communication systems

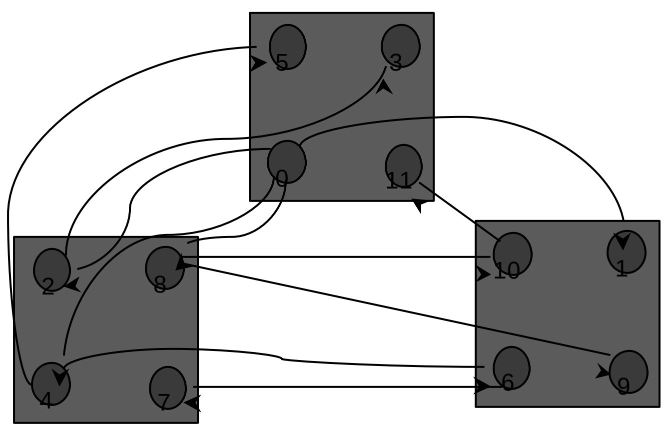
- Order(s) of magnitude performance differences
  - Latency
  - Bandwidth
- Examples
  - SMP Clusters
  - Computational grid

Topology unaware use



- Best case

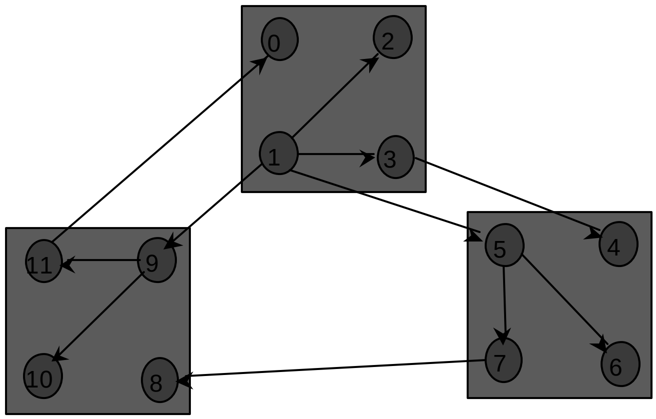
Topology unaware problem



- Worst case (*all* links slow)

LANS

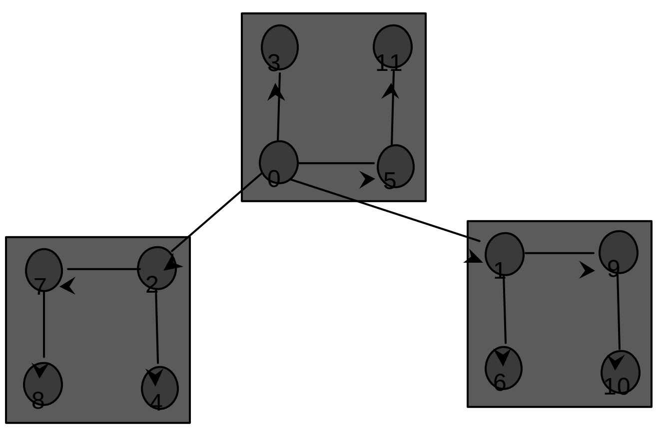
Topology unaware with non-zero root



- Common case (many links slow)

LANS

Topology aware solution



- Note that this is still not the best — see Tuesday's MPICH2 talk

Why a Multi-Level Implementation?

- Two levels do not capture all important systems
  - Where to split multilevel system into two?
    - Between two slowest levels?
    - Between two fastest levels?
    - Who determines?
- Original problem recurs at coalesced levels
- Two level approaches can degenerate to topology unaware solution for derived communicators

MPICH-G2 Topology Aware Implementation

- Determine topology
  - During start-up or communicator creation
  - Hidden communicators
    - Clusters
    - Masters
- Perform operation "recursively" over masters
- MPICH ADI *additions*
  - MPID_Depth
  - MPID_Clusterid

Hidden Communicator Creation

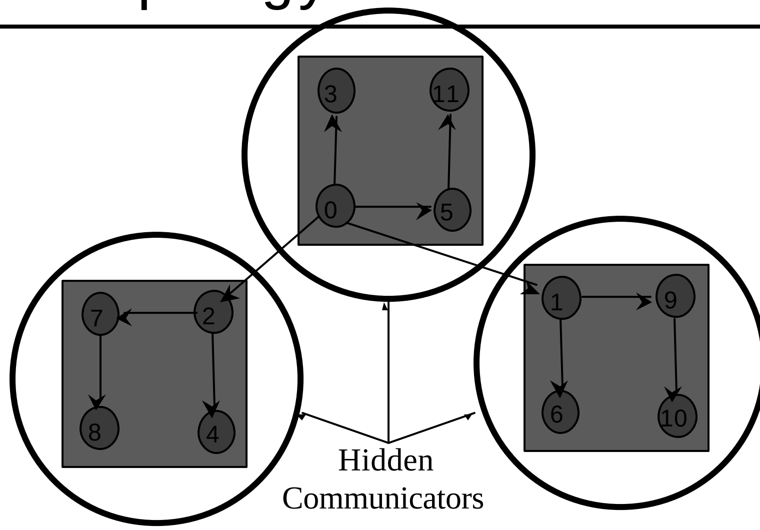
```
cluster[0] = copy of user level communicator;
for (i = 1; i < MPID_Depth; i++)
    MPI_Comm_split (cluster[i-1],
                        MPID_Clusterid, 0, &cluster[i]);
cluster_rank = 0;
for (i = MPID_Depth - 1; i > 0; i--)
    MPI_Comm_split (cluster[i-1],
                        cluster_rank, 0, &master[i]);
    if (cluster_rank != 0)
        MPI_Comm_free (master[i]);
    MPI_Comm_rank (cluster[i], &cluster_rank);
```


Broadcast algorithm

```
MPI_Bcast (buffer, count, datatype, 0, comm)
    for (i = 1; i < MPID_Depth; i++)
        MPI_Comm_rank (cluster[i], &cluster_rank);
        if (cluster_rank == 0)
          MPIR_LevelBcast (buffer, count,
                        datatype, 0, master[i], i);
```

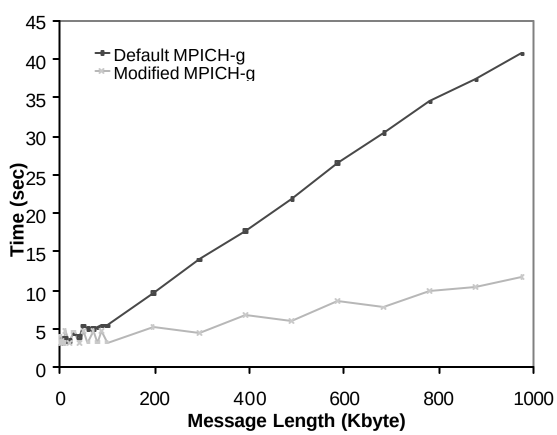
Non-zero roots:
- Substitute root for its master at faster levels
- Replace 0 with root in level broadcast if necessary

Topology aware solution



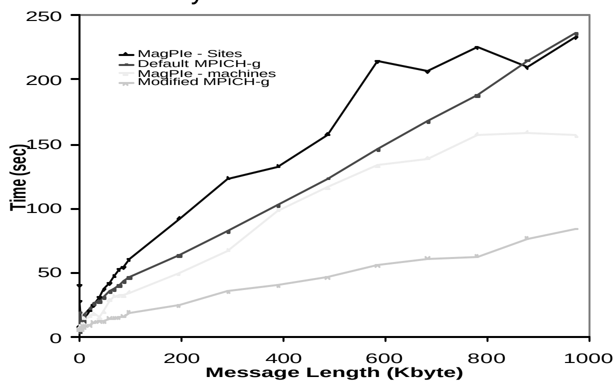
LAN Results

- 16 tasks each on ANL SP2 and ANL O2K



Comparative Performance Results

- 16 tasks each on ANL SP2, O2K and SDSC SP2
  - Three-level system



Topology Information

- MPICH-G2 exports information on system topology to applications programs through attributes:
- MPICHX_TOPOLOGY_DEPTHS
  - Vector, i^{th} value is depth of i^{th} process
- MPICHX_TOPOLOGY_COLORS
  - Vector of pointers to arrays; the i^{th} vector has length corresponding to the depth of that process and the values are the color at that level

Accessing Topology Information in MPICH-G2

```
int main (int argc, char *argv[])
{
   ...
   int *depths, **colors;
   ...
   rv = MPI_Attr_get(MPI_COMM_WORLD,
                     MPICHX_TOPOLOGY_DEPTHS, &depths, &flag1);
   rv = MPI_Attr_get(MPI_COMM_WORLD,
                     MPICHX_TOPOLOGY_COLORS, &colors, &flag2);
   if (flag1 && flag2) {
      /* depths[i] is depth of ith process */
      /* colors[i][j] is color of the ith process at the jth level */
     ...
   }
```

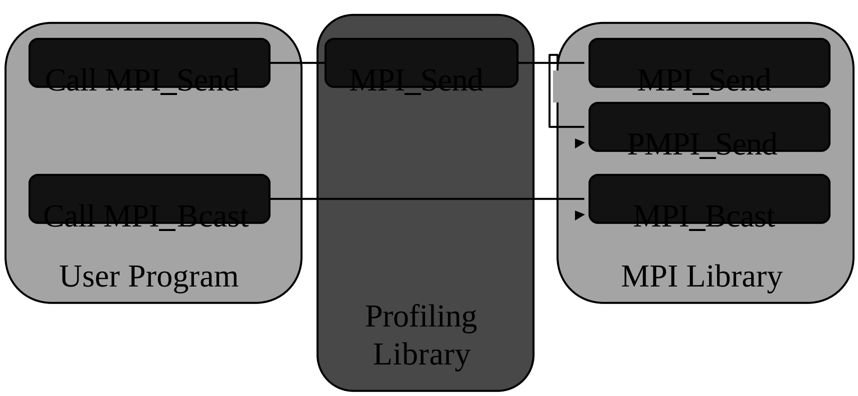

Performance and Debugging Tools

- Not a pretty picture
- No real grid debuggers
- Few application-level performance tools
- MPI provides a powerful hook on which to build customized performance and correctness debugging tools

Using PMPI routines

- PMPI allows selective replacement of MPI routines at link time (no need to recompile)
- Some libraries already make use of PMPI
- Some MPI implementations have PMPI bugs
    - PMPI may be in a separate library that some installations have not installed

Profiling Interface



Using the Profiling Interface

```
static int nsend = 0;

int MPI_Send( void *start, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm )
{
    nsend++;
    return PMPI_send(start, count, datatype,
                     dest, tag, comm);
}
```


Collecting Data From the Profiling Interface

- Use MPI_Finalize to force each process to either collect data (using MPI communication) or write data to local files. Then call PMPI_Finalize

Logging and Visualization Tools

- Upshot, Jumpshot, and MPE tools
  http://www.mcs.anl.gov/mpi/mpich
- Pallas VAMPIR
  http://www.pallas.com/
- Pablo http://www-pablo.cs.uiuc.edu/Projects/Pablo/pablo.html
- Paragraph
  http://www.ncsa.uiuc.edu/Apps/MCS/ParaGraph/ParaGraph.html
- Paradyn http://www.cs.wisc.edu/~paradyn
- Many other vendor tools exist
  - e.g., xmpi (SGI and HP)

Future Opportunities in MPI Implementations

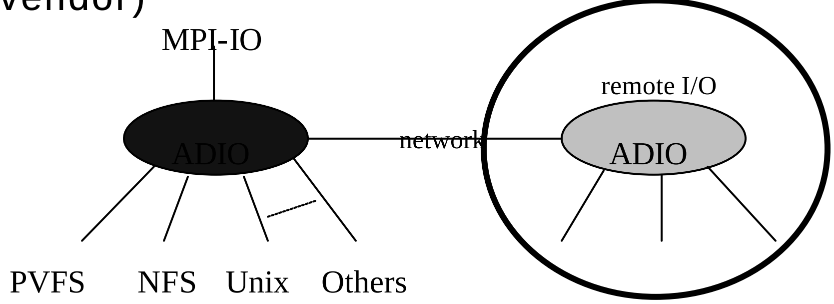
- I/O
  - Exploit MPI's sensible I/O semantics to get precise and latency tolerant behavior
- RMA
  - One-sided operations allow eager/ready-send behavior for messages of all sizes
- Dynamic processes
  - Major problem is the interaction with grid resource schedulers
- WAN Bandwidth
  - Multiple TCP paths (like GridFTP)
  - Customized UDP
    - May provide better congestion control, responsible sharing of bandwidth

A Few Comments on I/O

- Applications with data at one location and compute resources at another may become a more common class of grid codes
- POSIX I/O requires very strong coherency
  - So strong that many systems don’t provide POSIX semantics and instead provide ill-defined, cache-incoherent strategies
- MPI I/O has more precisely defined semantics that allow the MPI application to manage I/O sensibly (at least for a running MPI code)

ROMIO -- A Portable Implementation of MPI-IO

- Implementation strategy: an abstract device for I/O (ADIO)
- Tested for low overhead
- Can use any MPI implementation (MPICH, vendor)



Two-Phase Collective I/O

- ROMIO has an optimized implementation of two-phase collective I/O
- I/O is done in two phases: an I/O phase and a communication phase
- In the I/O phase, data is read/written in large chunks to minimize I/O latency
- Message-passing among compute nodes is used to redistribute data as needed

Current State of MPI I/O

- Only prototypes exist for grid I/O
- On the other hand, very efficient cluster and MPP implementations exist
  - Short term recommendation
    - Use MPI I/O within a cluster and MPI communication to move data on the Grid
  - Long term
    - Expect (or contribute to!) the development of MPI I/O for the grid

Fault Tolerance in MPI

- Can MPI be fault tolerant?
  - What does that mean?
- Implementation vs. Specification
  - Work to be done on the implementations
  - Work to be done on the algorithms
    - Semantically meaningful and efficient collective operations
  - Use MPI at the correct level
    - Build libraries to encapsulate important programming paradigms
- (Following slides are joint work with Rusty Lusk)

Myths and Facts

Myth: MPI behavior is defined by its implementations.
Fact: MPI behavior is defined by the Standard Document at http://www.mpi-forum.org

Myth: MPI is not fault tolerant.
Fact: This statement is not well formed. Its truth depends on what it means, and one can't tell from the statement itself. More later.

Myth: All processes of MPI programs exit if any one process crashes.
Fact: Sometimes they do; sometimes they don't; sometimes they should; sometimes they shouldn't. More later.

Myth: Fault tolerance means reliability.
Fact: These are completely different. Again, definitions are required.

More Myths and Facts

`Myth:` Fault tolerance is independent of performance.

`Fact:` In general, no. Perhaps for some (weak) aspects, yes. Support for fault tolerance will negatively impact performance.

`Myth:` Fault tolerance is a property of the MPI standard (which it doesn't have.

`Fact:` Fault tolerance is not a property of the specification, so it can't not have it. ☺

`Myth:` Fault tolerance is a property of an MPI implementation (which most don't have).

`Fact:` Fault tolerance is a property of a program. Some implementations make it easier to write fault-tolerant programs than others do.

What is Fault Tolerance Anyway?

- A fault-tolerant program can "survive" (in some sense we need to discuss) a failure of the infrastructure (machine crash, network failure, etc.)
- This is not in general completely attainable. (What if *all* processes crash?)
- How much is recoverable depends on how much *state* the failed component holds at the time of the crash.
  - In many master-slave algorithms a slave holds a small amount of easily recoverable state (the most recent subproblem it received).
  - In most mesh algorithms a process may hold a large amount of difficult-to-recover state (data values for some portion of the grid/matrix).
  - Communication networks hold varying amount of state in communication buffers.

LANS

What Does the Standard Say About Errors?

- A set of errors is defined, to be returned by MPI functions if MPI_ERRORS_RETURN is set.
- Implementations are allowed to extend this set.
- It is not required that subsequent operations work after an error is returned. (Or that they fail, either.)
- It may not be possible for an implementation to recover from some kinds of errors even enough to return an error code (and such implementations are conforming).
- Much is left to the implementation; some conforming implementations may return errors in situations where other conforming implementations abort. (See "quality of implementation" issue in the Standard.)
  - Implementations are allowed to trade performance against fault tolerance to meet the needs of their users

LANS

Some Approaches to Fault Tolerance in MPI Programs

- Master-slave algorithms using intercommunicators
  - No change to existing MPI semantics
  - MPI intercommunicators generalize the well-understood two party model to groups of processes, allowing either the master or slave to be a parallel program optimized for performance.
- Checkpointing
  - In wide use now
  - Plain vs. fancy
  - MPI-IO can help make it efficient
- Extending MPI with some new objects in order to allow a wider class of fault-tolerant programs.
  - The "pseudo-communicator"
- Another approach: Change semantics of existing MPI functions
  - No longer MPI (semantics, not syntax, defines MPI)

A Fault-Tolerant MPI Master/Slave Program

- Master process comes up alone first.
  - Size of MPI_COMM_WORLD = 1
- It creates slaves with MPI_Comm_spawn
  - Gets back an intercommunicator for each one
  - Sets MPI_ERRORS_RETURN on each
- Master communicates with each slave using its particular communicator
  - MPI_Send/Recv to/from rank 0 in remote group
  - Master maintains state information to restart each subproblem in case of failure
- Master may start replacement slave with MPI_Comm_spawn
- Slaves may themselves be parallel
  - Size of MPI_COMM_WORLD > 1 on slaves
  - Allows programmer to control tradeoff between fault tolerance and performance

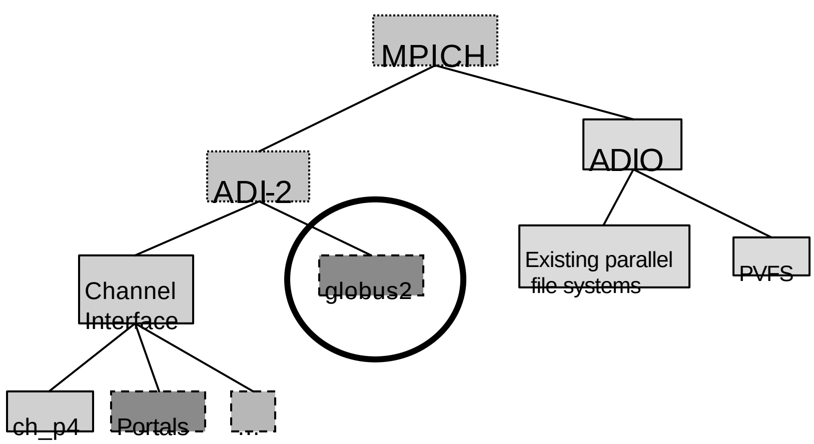
State of Fault Tolerance

- Few MPI implementations are robust in the presence of communication failures (LAM/MPI can survive some)
- This should change in the next year

MPI Implementations for the Grid

- Use any cluster-based implementation
  - Rely on ssh or independently started, implementation specific demons to start processes
  - Issues are
    - Executable distribution
    - Security
- Use IMPI
  - Only a few implementations
  - Simple security model
- Use an MPI implementation built on top of a solid Grid infrastructure
  - MPICH-G2

Structure of MPICH



LANS

What is MPICH-G2?

- Full implementation MPI 1.2 standard
- Developed by Nick Karonis (Northern Illinois University) and Brian Toonen (Argonne National Laboratory)
- MPICH-based, globus2 device
- Makes extensive use of Globus services, and therefore ...
- MPICH-G2 is a *grid-enabled* MPI

LANS

Globus services in MPICH-G2

- Launching application
  - Resource Specification Language (RSL)
  - The Dynamically-Updated Request Online Coallocator (DUROC)
  - Globus Resource Allocation Manager (GRAM)
  - globusrun
  - Globus Security Infrastructure (GSI)
- Staging
  - Globus Access to Secondary Storage (GASS)
- TCP Messaging
  - Globus I/O
  - Data Conversion

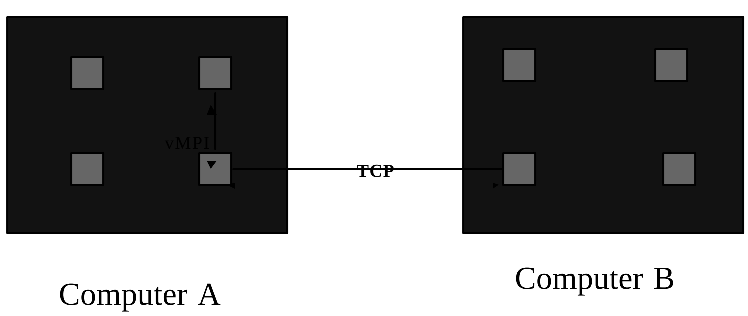
LANS

MPICH-G2 is Topology Aware

- Topology-aware collective operations
- Topology-discovery mechanisms
- Topology-aware multimethod messaging

LANS

Multimethod Support



When should MPICH-G2 be used?

- Applications that are distributed *by design*
  - Scientific applications that need either more compute power, more memory, or both
- Applications that are distributed *by nature*
  - Remote visualization applications, client/server applications, etc.

How to install MPICH-G2?

- Step 1 – Install Globus
  - Acquire and install Globus v2.0 or later (http://www.globus.org).
  - Deploy a Globus gatekeeper (a demon) on each machine (not node!) you intend to run.
  - Acquire Globus identification (request from ca@globus.org) and set it up.
  - Add your Globus ID to Globus “gridmap” file on each machine you intend to run.
  - Test with “hello, world” program (from “Troubleshooting” section of www.globus.org/mpi).

How to install MPICH-G2? (cont.)

- Step 2 – Install MPICH-G2
  - Acquire MPICH v1.2.4 or later.
  - setenv GLOBUS_LOCATION to your Globus installation.
  - Pick a Globus “flavor” (never pick “threaded” flavor, always pick “mpi” flavor where available).
  - Configure MPICH with -device=globus2, make, make install

How to use MPICH-G2?

- Step 1 – Compiling your MPI application
  - source the file $GLOBUS_LOCATION/etc/globus-user-env.csh
  - Use MPICH-G2 compiler/linker:
    - <mpichpath>bin/mpicc
    - <mpichpath>bin/mpiCC
    - <mpichpath>bin/mpif77
    - <mpichpath>bin/mpif90

How to use MPICH-G2? (cont)

- Step 2 – Running your MPI application
  - Use mpirun as described in manual, e.g.,

```
% mpirun –np 2 a.out arg1 arg2
```

Or

- Write your own Globus RSL script (www.globus.org) and supply that ***only***

```
% mpirun –globusrsl myfile.rsl
```


Optional Execution-time Specifications

- Setting IP address range to specify a network interface
  - setenv MPICH_GLOBUS2_USE_NETWORK_INTERFACE <ipaddr>
- Setting TCP port range
  - setenv GLOBUS_TCP_PORT_RANGE “min max”
- *Request* TCP buffer size
  - setenv MPICH_GLOBUS2_TCP_BUFFER_SIZE nbytes

MPICH-G2, Globus, and Firewalls

- It is possible to run MPICH-G2 applications *through* firewalls, but it takes sys admin cooperation.
- Described briefly, sys admins creates a small “hole” in the firewall called a *controllable ephemeral port*.
- You use GLOBUS_TCP_PORT_RANGE to specify that port.
- For full dicussion of Globus and firewalls, see http://www.globus.org/security/v2.0/firewalls.html

Server Example

```
#include <stdio.h>
#include " mpi.h"

int main(int argc, char **argv)
{
   int passed_num, my_id;
   char port_name[MPI_MAX_PORT_NAME];
   MPI_Comm newcomm;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

   passed_num = 111;
```

Server Example (con't)

```
    if (my_id == 0)    {
        MPI_Open_port(MPI_INFO_NULL, port_name);
        printf("%s\n\n", port_name); fflush(stdout);
    }

    MPI_Comm_accept(port_name, MPI_INFO_NULL, 0,
                    MPI_COMM_WORLD, &newcomm);

    if (my_id == 0)  {
            MPI_Send(&passed_num, 1, MPI_INT, 0, 0, newcomm);
            printf("after sending passed_num %d\n", passed_num);
    fflush(stdout);
            MPI_Close_port(port_name);
    }
    MPI_Finalize();
    return 0;
}
```


Client Example

```
#include <stdio.h>
#include " mpi.h"

int main(int argc, char **argv)
{
    int passed_num, my_id;
    MPI_Comm newcomm;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

    MPI_Comm_connect(argv[1], MPI_INFO_NULL, 0,
                     MPI_COMM_WORLD, &newcomm);
```

Client Example (con't)

```
  if (my_id == 0) {
  MPI_Status status;
  MPI_Recv(&passed_num, 1, MPI_INT, 0, 0, newcomm,
              &status);
      printf("after receiving passed_num %d\n", passed_num);
      fflush(stdout);
  }

  MPI_Finalize();
  return 0;
}
```


Conclusions

- MPI the specification provides a good programming model for the Grid
- MPI implementations are usable, but more needs to be done
  - MPICH-G2: www.globus.org/mpi
- Many opportunities for both using MPI on the Grid and contributing to developing implementations that are "grid friendly"